Laravel 9 · Коллекции

- Введение
  - Создание коллекций
  - Расширение коллекций
- Доступные методы
- Сообщения высшего порядка
- Отложенные коллекции
  - Введение в отложенные коллекции
  - Создание отложенных коллекций
  - Контракт `Enumerable`
  - Методы отложенных коллекций

Введение

Класс `Illuminate\Support\Collection` обеспечивает гибкую и удобную обертку для работы с массивами данных. Например, посмотрите на следующий код. Здесь мы будем использовать помощник `collect`, чтобы создать новый экземпляр коллекции из массива, запустим функцию `strtoupper` для каждого элемента, а затем удалим все пустые элементы:

```
$collection = collect(['taylor', 'abigail', null])->map(function ($name) {
    return strtoupper($name);
})->reject(function ($name) {
    return empty($name);
});
```

Как видите, класс `Collection` позволяет объединять необходимые вам методы в цепочку для выполнения последовательного перебора и сокращения базового массива. В-основном коллекции неизменяемы, то есть каждый метод коллекции возвращает совершенно новый экземпляр `Collection` .

Создание коллекций

Как упоминалось выше, помощник `collect` возвращает новый экземпляр `Illuminate\Support\Collection` для переданного массива. Итак, создать коллекцию очень просто:

```
$collection = collect([1, 2, 3]);
```

> {tip} Результаты запросов Eloquent всегда возвращаются как экземпляры `Collection` .

Расширение коллекций

Класс `Collection` являются «макропрограммируемым», что позволяет вам добавлять дополнительные методы к классу во время выполнения. Метод `macro` класса `Illuminate\Support\Collection` принимает замыкание, которое будет выполнено при вызове вашей макрокоманды. Замыкание макрокоманды может обращаться к другим методам коллекции через `$this` , как если бы это был реальный метод класса коллекции. Например, следующий код добавляет метод `toUpper` классу `Collection` :

```
use Illuminate\Support\Collection;
use Illuminate\Support\Str;

Collection::macro('toUpper', function () {
    return $this->map(function ($value) {
        return Str::upper($value);
    });
});

$collection = collect(['first', 'second']);

$upper = $collection->toUpper();

// ['FIRST', 'SECOND']
```

Как правило, макрокоманды коллекций должны быть объявлены в методе `boot` поставщика служб.

Аргументы макрокоманды

При необходимости вы можете определить макрокоманды, которые принимают дополнительные аргументы:

```
use Illuminate\Support\Collection;
use Illuminate\Support\Facades\Lang;
```

```
Collection::macro('toLocale', function ($locale) {
    return $this->map(function ($value) use ($locale) {
        return Lang::get($value, [], $locale);
    });
});

$collection = collect(['first', 'second']);

$translated = $collection->toLocale('es');
```

Доступные методы

В большей части оставшейся документации по коллекциям мы обсудим каждый метод, доступный в классе `Collection`. Помните, что все эти методы можно объединить в цепочку для последовательного управления базовым массивом. Более того, почти каждый метод возвращает новый экземпляр `Collection`, позволяя вам при необходимости сохранить исходную копию коллекции:

- all
- average
- avg
- chunk
- chunkWhile
- collapse
- collect
- combine
- concat
- contains
- containsStrict
- count
- countBy
- crossJoin
- dd
- diff
- diffAssoc
- diffKeys
- doesntContain
- dump
- duplicates
- duplicatesStrict
- each
- eachSpread

- every
- except
- filter
- first
- firstOrFail
- firstWhere
- flatMap
- flatten
- flip
- forget
- forPage
- get
- groupBy
- has
- implode
- intersect
- intersectByKeys
- isEmpty
- isNotEmpty
- join
- keyBy
- keys
- last
- lazy
- macro
- make
- map
- mapInto
- mapSpread
- mapToGroups
- mapWithKeys
- max
- median
- merge
- mergeRecursive
- min
- mode
- nth
- only
- pad

- partition
- pipe
- pipeInto
- pipeThrough
- pluck
- pop
- prepend
- pull
- push
- put
- random
- range
- reduce
- reduceSpread
- reject
- replace
- replaceRecursive
- reverse
- search
- shift
- shuffle
- skip
- skipUntil
- skipWhile
- slice
- sliding
- sole
- some
- sort
- sortBy
- sortByDesc
- sortDesc
- sortKeys
- sortKeysDesc
- sortKeysUsing
- splice
- split
- splitIn
- sum
- take

- takeUntil
- takeWhile
- tap
- times
- toArray
- toJson
- transform
- undot
- union
- unique
- uniqueStrict
- unless
- unlessEmpty
- unlessNotEmpty
- unwrap
- value
- values
- when
- whenEmpty
- whenNotEmpty
- where
- whereStrict
- whereBetween
- whereIn
- whereInStrict
- whereInstanceOf
- whereNotBetween
- whereNotIn
- whereNotInStrict
- whereNotNull
- whereNull
- wrap
- zip

Список методов

`all()`

Метод `all` возвращает базовый массив, представленный коллекцией:

```
collect([1, 2, 3])->all();

// [1, 2, 3]
```

average()

Псевдоним для метода `avg`.

avg()

Метод `avg` возвращает среднее значение переданного ключа:

```
$average = collect([
    ['foo' => 10],
    ['foo' => 10],
    ['foo' => 20],
    ['foo' => 40]
])->avg('foo');

// 20

$average = collect([1, 1, 2, 4])->avg();

// 2
```

chunk()

Метод `chunk` разбивает коллекцию на несколько меньших коллекций указанного размера:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7]);

$chunks = $collection->chunk(4);

$chunks->all();

// [[1, 2, 3, 4], [5, 6, 7]]
```

Этот метод особенно полезен в шаблонах при работе с сеткой, такой как Bootstrap. Например, представьте, что у вас есть коллекция моделей Eloquent, которые вы хотите отобразить в сетке:

```
@foreach ($products->chunk(3) as $chunk)
    <div class="row">
        @foreach ($chunk as $product)
            <div class="col-xs-4">{{ $product->name }}</div>
        @endforeach
    </div>
@endforeach
```

chunkWhile()

Метод `chunkWhile` разбивает коллекцию на несколько меньших по размеру коллекций на основе результата переданного замыкания. Переменная `$chunk`, переданная в замыкание, может использоваться для проверки предыдущего элемента:

```
$collection = collect(str_split('AABBCCCD'));

$chunks = $collection->chunkWhile(function ($value, $key, $chunk) {
    return $value === $chunk->last();
});

$chunks->all();

// [['A', 'A'], ['B', 'B'], ['C', 'C', 'C'], ['D']]
```

collapse()

Метод `collapse` сворачивает коллекцию массивов в единую плоскую коллекцию:

```
$collection = collect([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
]);

$collapsed = $collection->collapse();

$collapsed->all();

// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

collect()

Метод `collect` возвращает новый экземпляр `Collection` с элементами, находящимися в текущей коллекции:

```
$collectionA = collect([1, 2, 3]);

$collectionB = $collectionA->collect();

$collectionB->all();

// [1, 2, 3]
```

Метод `collect` в первую очередь полезен для преобразования отложенных коллекций в стандартные экземпляры `Collection`:

```
$lazyCollection = LazyCollection::make(function () {
    yield 1;
    yield 2;
    yield 3;
});

$collection = $lazyCollection->collect();

get_class($collection);

// 'Illuminate\Support\Collection'

$collection->all();

// [1, 2, 3]
```

> {tip} Метод `collect` особенно полезен, когда у вас есть экземпляр `Enumerable` и вам нужен «не-отложенный» экземпляр коллекции. Так как `collect()` является частью контракта `Enumerable`, вы можете безопасно использовать его для получения экземпляра `Collection`.

combine()

Метод `combine` объединяет значения коллекции в качестве ключей со значениями другого массива или коллекции:

```
$collection = collect(['name', 'age']);

$combined = $collection->combine(['George', 29]);

$combined->all();

// ['name' => 'George', 'age' => 29]
```

concat()

Метод `concat` добавляет значения переданного массива или коллекции в конец другой коллекции:

```
$collection = collect(['John Doe']);

$concatenated = $collection->concat(['Jane Doe'])->concat(['name' => 'Johnny Doe']);

$concatenated->all();

// ['John Doe', 'Jane Doe', 'Johnny Doe']
```

Метод `concat` численно переиндексирует ключи элементов, добавленных в исходную коллекцию. Чтобы сохранить ключи в ассоциативных коллекциях, см. метод merge.

`contains()`

Метод `contains` определяет присутствие переданного элемента в коллекции. Вы также можете передать замыкание методу `contains`, чтобы определить присутствие элемента с указанным критерием истинности в коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->contains(function ($value, $key) {
    return $value > 5;
});

// false
```

Как вариант, вы можете передать строку методу `contains`, чтобы определить присутствие элемента с указанным значением в коллекции:

```
$collection = collect(['name' => 'Desk', 'price' => 100]);

$collection->contains('Desk');

// true

$collection->contains('New York');

// false
```

Вы также можете передать пару ключ / значение методу `contains`, который определит присутствие переданной пары в коллекции:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
]);

$collection->contains('product', 'Bookcase');

// false
```

Метод `contains` использует «гибкое» сравнение при проверке значений элементов, то есть строка с целочисленным значением будет считаться равной целому числу того же значения. Используйте метод `containsStrict` для фильтрации с использованием «жесткого» сравнения.

Противоположным методу `contains` является метод doesntContain.

`containsStrict()`

Этот метод имеет ту же сигнатуру, что и метод `contains`; однако, все значения сравниваются с использованием «жесткого» сравнения.

> {tip} Поведение этого метода изменяется при использовании коллекций Eloquent.

count()

Метод `count` возвращает общее количество элементов в коллекции:

```
$collection = collect([1, 2, 3, 4]);

$collection->count();

// 4
```

countBy()

Метод `countBy` подсчитывает вхождения значений в коллекцию. По умолчанию метод подсчитывает вхождения каждого элемента, что позволяет подсчитать определенные «типы» элементов в коллекции:

```
$collection = collect([1, 2, 2, 2, 3]);

$counted = $collection->countBy();

$counted->all();

// [1 => 1, 2 => 3, 3 => 1]
```

Вы можете передать замыкание методу `countBy` для подсчета всех элементов по собственным критериям:

```
$collection = collect(['alice@gmail.com', 'bob@yahoo.com', 'carlos@gmail.com']);

$counted = $collection->countBy(function ($email) {
    return substr(strrchr($email, "@"), 1);
});

$counted->all();

// ['gmail.com' => 2, 'yahoo.com' => 1]
```

crossJoin()

Метод `crossJoin` перекрестно соединяет значения коллекции среди переданных массивов или коллекций, возвращая декартово произведение со всеми возможными перестановками:

```
$collection = collect([1, 2]);

$matrix = $collection->crossJoin(['a', 'b']);
```

```
$matrix->all();

/*
    [
        [1, 'a'],
        [1, 'b'],
        [2, 'a'],
        [2, 'b'],
    ]
*/

$collection = collect([1, 2]);

$matrix = $collection->crossJoin(['a', 'b'], ['I', 'II']);

$matrix->all();

/*
    [
        [1, 'a', 'I'],
        [1, 'a', 'II'],
        [1, 'b', 'I'],
        [1, 'b', 'II'],
        [2, 'a', 'I'],
        [2, 'a', 'II'],
        [2, 'b', 'I'],
        [2, 'b', 'II'],
    ]
*/
```

dd()

Метод `dd` выводит элементы коллекции и завершает выполнение скрипта:

```
$collection = collect(['John Doe', 'Jane Doe']);

$collection->dd();

/*
    Collection {
        #items: array:2 [
            0 => "John Doe"
            1 => "Jane Doe"
        ]
    }
*/
```

Если вы не хотите останавливать выполнение вашего скрипта, используйте вместо этого метод `dump`.

diff()

Метод `diff` сравнивает коллекцию с другой коллекцией или простым массивом PHP на основе его значений. Этот метод вернет значения из исходной коллекции, которых нет в переданной коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);

$diff = $collection->diff([2, 4, 6, 8]);

$diff->all();

// [1, 3, 5]
```

> {tip} Поведение этого метода изменяется при использовании коллекций Eloquent.

diffAssoc()

Метод `diffAssoc` сравнивает коллекцию с другой коллекцией или простым массивом PHP на основе его ключей и значений. Этот метод вернет пары ключ / значение из исходной коллекции, которых нет в переданной коллекции:

```
$collection = collect([
    'color' => 'orange',
    'type' => 'fruit',
    'remain' => 6,
]);

$diff = $collection->diffAssoc([
    'color' => 'yellow',
    'type' => 'fruit',
    'remain' => 3,
    'used' => 6,
]);

$diff->all();

// ['color' => 'orange', 'remain' => 6]
```

diffKeys()

Метод `diffKeys` сравнивает коллекцию с другой коллекцией или простым массивом PHP на основе его ключей. Этот метод вернет пары ключ / значение из исходной коллекции, которых нет в переданной коллекции:

```
$collection = collect([
    'one' => 10,
    'two' => 20,
    'three' => 30,
    'four' => 40,
    'five' => 50,
]);
```

```
$diff = $collection->diffKeys([
    'two' => 2,
    'four' => 4,
    'six' => 6,
    'eight' => 8,
]);

$diff->all();

// ['one' => 10, 'three' => 30, 'five' => 50]
```

doesntContain()

Метод `doesntContain` определяет отсутствие переданного элемента в коллекции. Вы также можете передать замыкание методу `doesntContain`, чтобы определить отсутствие элемента с указанным критерием истинности в коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->doesntContain(function ($value, $key) {
    return $value < 5;
});

// false
```

Как вариант, вы можете передать строку методу `doesntContain`, чтобы определить отсутствие элемента с указанным значением в коллекции:

```
$collection = collect(['name' => 'Desk', 'price' => 100]);

$collection->doesntContain('Table');

// true

$collection->doesntContain('Desk');

// false
```

Вы также можете передать пару ключ / значение методу `doesntContain`, который определит отсутствие переданной пары в коллекции:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
]);

$collection->doesntContain('product', 'Bookcase');

// true
```

Метод `doesntContain` использует «гибкое» сравнение при проверке значений элементов, то есть строка с целочисленным значением будет считаться равной целому числу того же значения.

dump()

Метод `dump` выводит элементы коллекции:

```
$collection = collect(['John Doe', 'Jane Doe']);

$collection->dump();

/*
    Collection {
        #items: array:2 [
            0 => "John Doe"
            1 => "Jane Doe"
        ]
    }
*/
```

Если вы хотите прекратить выполнение скрипта после вывода элементов коллекции, используйте вместо этого метод `dd` .

duplicates()

Метод `duplicates` извлекает и возвращает повторяющиеся значения из коллекции:

```
$collection = collect(['a', 'b', 'a', 'c', 'b']);

$collection->duplicates();

// [2 => 'a', 4 => 'b']
```

Если коллекция содержит массивы или объекты, вы можете передать ключ атрибутов, которые вы хотите проверить на наличие повторяющихся значений:

```
$employees = collect([
    ['email' => 'abigail@example.com', 'position' => 'Developer'],
    ['email' => 'james@example.com', 'position' => 'Designer'],
    ['email' => 'victoria@example.com', 'position' => 'Developer'],
]);

$employees->duplicates('position');

// [2 => 'Developer']
```

duplicatesStrict()

Этот метод имеет ту же сигнатуру, что и метод `duplicates` ; однако, все значения сравниваются с использованием «жесткого» сравнения.

each()

Метод `each` перебирает элементы в коллекции и передает каждый элемент в замыкание:

```
$collection->each(function ($item, $key) {
    //
});
```

Если вы хотите прекратить итерацию по элементам, вы можете вернуть `false` из вашего замыкания:

```
$collection->each(function ($item, $key) {
    if (/* condition */) {
        return false;
    }
});
```

eachSpread()

Метод `eachSpread` выполняет итерацию по элементам коллекции, передавая значение каждого вложенного элемента в замыкание:

```
$collection = collect([['John Doe', 35], ['Jane Doe', 33]]);

$collection->eachSpread(function ($name, $age) {
    //
});
```

Если вы хотите прекратить итерацию по элементам, вы можете вернуть `false` из вашего замыкания:

```
$collection->eachSpread(function ($name, $age) {
    return false;
});
```

every()

Метод `every` используется для проверки того, что все элементы коллекции проходят указанный тест истинности:

```
collect([1, 2, 3, 4])->every(function ($value, $key) {
    return $value > 2;
});
```

```
// false
```

Если коллекция пуста, метод `every` вернет `true`:

```
$collection = collect([]);

$collection->every(function ($value, $key) {
    return $value > 2;
});

// true
```

except()

Метод `except` возвращает все элементы из коллекции, кроме тех, которые имеют указанные ключи:

```
$collection = collect(['product_id' => 1, 'price' => 100, 'discount' => false]);

$filtered = $collection->except(['price', 'discount']);

$filtered->all();

// ['product_id' => 1]
```

Противоположным методу `except` является метод only.

> {tip} Поведение этого метода изменяется при использовании коллекций Eloquent.

filter()

Метод `filter` фильтрует коллекцию, используя переданное замыкание, сохраняя только те элементы, которые проходят указанный тест истинности:

```
$collection = collect([1, 2, 3, 4]);

$filtered = $collection->filter(function ($value, $key) {
    return $value > 2;
});

$filtered->all();

// [3, 4]
```

Если замыкание не указано, то все записи коллекции, эквивалентные `false`, будут удалены:

```
$collection = collect([1, 2, 3, null, false, '', 0, []]);
```

```
$collection->filter()->all();

// [1, 2, 3]
```

Противоположным методу `filter` является метод reject.

first()

Метод `first` возвращает первый элемент из коллекции, который проходит указанную проверку истинности:

```
collect([1, 2, 3, 4])->first(function ($value, $key) {
    return $value > 2;
});

// 3
```

Вы также можете вызвать метод `first` без аргументов, чтобы получить первый элемент из коллекции. Если коллекция пуста, возвращается `null`:

```
collect([1, 2, 3, 4])->first();

// 1
```

firstOrFail()

Метод `firstOrFail` идентичен методу `first`; однако, если результат не найден, то будет выброшено исключение `Illuminate\Support\ItemNotFoundException`:

```
collect([1, 2, 3, 4])->firstOrFail(function ($value, $key) {
    return $value > 5;
});

// Выброшено исключение ItemNotFoundException ...
```

Вы также можете вызвать метод `firstOrFail` без аргументов, чтобы получить первый элемент коллекции. Если коллекция пуста, то будет выброшено исключение `Illuminate\Support\ItemNotFoundException`:

```
collect([])->firstOrFail();

// Выброшено исключение ItemNotFoundException ...
```

firstWhere()

Метод `firstWhere` возвращает первый элемент коллекции с переданной парой ключ / значение:

```
$collection = collect([
    ['name' => 'Regena', 'age' => null],
    ['name' => 'Linda', 'age' => 14],
    ['name' => 'Diego', 'age' => 23],
    ['name' => 'Linda', 'age' => 84],
]);

$collection->firstWhere('name', 'Linda');

// ['name' => 'Linda', 'age' => 14]
```

Вы также можете вызвать метод `firstWhere` с оператором сравнения:

```
$collection->firstWhere('age', '>=', 18);

// ['name' => 'Diego', 'age' => 23]
```

Подобно методу where, вы можете передать один аргумент методу `firstWhere` . В этом сценарии метод `firstWhere` вернет первый элемент, для которого значение данного ключа элемента является «истинным»:

```
$collection->firstWhere('age');

// ['name' => 'Linda', 'age' => 14]
```

`flatMap()`

Метод `flatMap` выполняет итерацию по коллекции и передает каждое значение переданному замыканию. Замыкание может изменить элемент и вернуть его, таким образом формируя новую коллекцию измененных элементов. Затем массив преобразуется в плоскую структуру:

```
$collection = collect([
    ['name' => 'Sally'],
    ['school' => 'Arkansas'],
    ['age' => 28]
]);

$flattened = $collection->flatMap(function ($values) {
    return array_map('strtoupper', $values);
});

$flattened->all();

// ['name' => 'SALLY', 'school' => 'ARKANSAS', 'age' => '28'];
```

flatten()

Метод `flatten` объединяет многомерную коллекцию в одноуровневую:

```
$collection = collect([
    'name' => 'taylor',
    'languages' => [
        'php', 'javascript'
    ]
]);

$flattened = $collection->flatten();

$flattened->all();

// ['taylor', 'php', 'javascript'];
```

Если необходимо, вы можете передать методу `flatten` аргумент «глубины»:

```
$collection = collect([
    'Apple' => [
        [
            'name' => 'iPhone 6S',
            'brand' => 'Apple'
        ],
    ],
    'Samsung' => [
        [
            'name' => 'Galaxy S7',
            'brand' => 'Samsung'
        ],
    ],
]);

$products = $collection->flatten(1);

$products->values()->all();

/*
    [
        ['name' => 'iPhone 6S', 'brand' => 'Apple'],
        ['name' => 'Galaxy S7', 'brand' => 'Samsung'],
    ]
*/
```

В этом примере вызов `flatten` без указания глубины также привел бы к сглаживанию вложенных массивов, что привело бы к `['iPhone 6S', 'Apple', 'Galaxy S7', 'Samsung']`. Предоставление глубины позволяет указать количество уровней, на которые будут сглажены вложенные массивы.

flip()

Метод `flip` меняет местами ключи коллекции на их соответствующие значения:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$flipped = $collection->flip();

$flipped->all();

// ['taylor' => 'name', 'laravel' => 'framework']
```

forget()

Метод `forget` удаляет элемент из коллекции по его ключу:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$collection->forget('name');

$collection->all();

// ['framework' => 'laravel']
```

> {note} В отличие от большинства других методов коллекции, `forget` модифицирует коллекцию.

forPage()

Метод `forPage` возвращает новую коллекцию, содержащую элементы, которые будут присутствовать на указанном номере страницы. Метод принимает номер страницы в качестве первого аргумента и количество элементов, отображаемых на странице, в качестве второго аргумента:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9]);

$chunk = $collection->forPage(2, 3);

$chunk->all();

// [4, 5, 6]
```

get()

Метод `get` возвращает элемент по указанному ключу. Если ключ не существует, возвращается `null` :

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('name');
```

```
// taylor
```

При желании вы можете передать значение по умолчанию в качестве второго аргумента:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('age', 34);

// 34
```

Вы даже можете передать замыкание как значение метода по умолчанию. Результат замыкания будет возвращен, если указанный ключ не существует:

```
$collection->get('email', function () {
    return 'taylor@example.com';
});

// taylor@example.com
```

groupBy()

Метод `groupBy` группирует элементы коллекции по указанному ключу:

```
$collection = collect([
    ['account_id' => 'account-x10', 'product' => 'Chair'],
    ['account_id' => 'account-x10', 'product' => 'Bookcase'],
    ['account_id' => 'account-x11', 'product' => 'Desk'],
]);

$grouped = $collection->groupBy('account_id');

$grouped->all();

/*
    [
        'account-x10' => [
            ['account_id' => 'account-x10', 'product' => 'Chair'],
            ['account_id' => 'account-x10', 'product' => 'Bookcase'],
        ],
        'account-x11' => [
            ['account_id' => 'account-x11', 'product' => 'Desk'],
        ],
    ]
*/
```

Вместо передачи строкового ключа вы можете передать замыкание. Замыкание должно вернуть значение, используемое в качестве ключа для группировки:

```
$grouped = $collection->groupBy(function ($item, $key) {
    return substr($item['account_id'], -3);
});

$grouped->all();

/*
    [
        'x10' => [
            ['account_id' => 'account-x10', 'product' => 'Chair'],
            ['account_id' => 'account-x10', 'product' => 'Bookcase'],
        ],
        'x11' => [
            ['account_id' => 'account-x11', 'product' => 'Desk'],
        ],
    ]
*/
```

В виде массива можно передать несколько критериев группировки. Каждый элемент массива будет применен к соответствующему уровню в многомерном массиве:

```
$data = new Collection([
    10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],
    20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],
    30 => ['user' => 3, 'skill' => 2, 'roles' => ['Role_1']],
    40 => ['user' => 4, 'skill' => 2, 'roles' => ['Role_2']],
]);

$result = $data->groupBy(['skill', function ($item) {
    return $item['roles'];
}], preserveKeys: true);

/*
[
    1 => [
        'Role_1' => [
            10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],
            20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],
        ],
        'Role_2' => [
            20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],
        ],
        'Role_3' => [
            10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],
        ],
    ],
    2 => [
        'Role_1' => [
            30 => ['user' => 3, 'skill' => 2, 'roles' => ['Role_1']],
        ],
        'Role_2' => [
            40 => ['user' => 4, 'skill' => 2, 'roles' => ['Role_2']],
        ],
    ],
```

```
];
*/
```

has()

Метод `has` определяет, существует ли переданный ключ в коллекции:

```
$collection = collect(['account_id' => 1, 'product' => 'Desk', 'amount' => 5]);

$collection->has('product');

// true

$collection->has(['product', 'amount']);

// true

$collection->has(['amount', 'price']);

// false
```

implode()

Метод `implode` объединяет элементы коллекции. Его аргументы зависят от типа элементов в коллекции. Если коллекция содержит массивы или объекты, то вы должны передать ключ объединяемых атрибутов и «связующую строку», размещаемую между значениями:

```
$collection = collect([
    ['account_id' => 1, 'product' => 'Desk'],
    ['account_id' => 2, 'product' => 'Chair'],
]);

$collection->implode('product', ', ');

// Desk, Chair
```

Если коллекция содержит простые строки или числовые значения, вы должны передать «связующую строку» как единственный аргумент методу:

```
collect([1, 2, 3, 4, 5])->implode('-');

// '1-2-3-4-5'
```

intersect()

Метод `intersect` удаляет любые значения из исходной коллекции, которых нет в указанном массиве или коллекции. Полученная коллекция сохранит ключи исходной коллекции:

```
$collection = collect(['Desk', 'Sofa', 'Chair']);

$intersect = $collection->intersect(['Desk', 'Chair', 'Bookcase']);

$intersect->all();

// [0 => 'Desk', 2 => 'Chair']
```

> {tip} Поведение этого метода изменяется при использовании коллекций Eloquent.

intersectByKeys()

Метод `intersectByKeys` удаляет все ключи и соответствующие им значения из исходной коллекции, ключи которых отсутствуют в указанном массиве или коллекции:

```
$collection = collect([
    'serial' => 'UX301', 'type' => 'screen', 'year' => 2009,
]);

$intersect = $collection->intersectByKeys([
    'reference' => 'UX404', 'type' => 'tab', 'year' => 2011,
]);

$intersect->all();

// ['type' => 'screen', 'year' => 2009]
```

isEmpty()

Метод `isEmpty` возвращает `true`, если коллекция пуста; в противном случае возвращается `false`:

```
collect([])->isEmpty();

// true
```

isNotEmpty()

Метод `isNotEmpty` возвращает `true`, если коллекция не пуста; в противном случае возвращается `false`:

```
collect([])->isNotEmpty();

// false
```

join()

Метод `join` объединяет значения коллекции в строку. Используя второй аргумент этого метода, вы также можете указать, как последний элемент должен быть добавлен к строке:

```
collect(['a', 'b', 'c'])->join(', '); // 'a, b, c'
collect(['a', 'b', 'c'])->join(', ', ', and '); // 'a, b, and c'
collect(['a', 'b'])->join(', ', ' and '); // 'a and b'
collect(['a'])->join(', ', ' and '); // 'a'
collect([])->join(', ', ' and '); // ''
```

keyBy()

Метод `keyBy` возвращает коллекцию, элементы которой будут образованы путем присвоения ключей элементам базовой коллекции. Если у нескольких элементов один и тот же ключ, в новой коллекции появится только последний:

```
$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'Desk'],
    ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$keyed = $collection->keyBy('product_id');

$keyed->all();

/*
    [
        'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
        'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
    ]
*/
```

Вы также можете передать методу замыкание. Замыкание должно возвращать имя для ключа коллекции:

```
$keyed = $collection->keyBy(function ($item) {
    return strtoupper($item['product_id']);
});

$keyed->all();

/*
    [
        'PROD-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
        'PROD-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
    ]
*/
```

keys()

Метод `keys` возвращает все ключи коллекции:

```
$collection = collect([
    'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$keys = $collection->keys();

$keys->all();

// ['prod-100', 'prod-200']
```

last()

Метод `last` возвращает последний элемент в коллекции, который проходит указанную проверку истинности:

```
collect([1, 2, 3, 4])->last(function ($value, $key) {
    return $value < 3;
});

// 2
```

Вы также можете вызвать метод `last` без аргументов, чтобы получить последний элемент коллекции. Если коллекция пуста, возвращается `null` :

```
collect([1, 2, 3, 4])->last();

// 4
```

lazy()

Метод `lazy` возвращает новый экземпляр `LazyCollection` из массива элементов:

```
$lazyCollection = collect([1, 2, 3, 4])->lazy();

get_class($lazyCollection);

// Illuminate\Support\LazyCollection

$lazyCollection->all();

// [1, 2, 3, 4]
```

Это особенно полезно, когда вам нужно выполнить преобразования «коллекции», содержащей много элементов:

```
$count = $hugeCollection
    ->lazy()
    ->where('country', 'FR')
    ->where('balance', '>', '100')
    ->count();
```

Преобразовывая коллекцию в `LazyCollection`, мы избегаем необходимости в выделении дополнительной памяти. Хотя исходная коллекция по-прежнему сохраняет *свои* значения в памяти, последующая фильтрация этого не делает. Поэтому при фильтрации результатов коллекции практически не выделяется дополнительная память.

`macro()`

Статический метод `macro` позволяет вам добавлять методы к классу `Collection` во время выполнения. Обратитесь к документации по расширению коллекций для получения дополнительной информации.

`make()`

Статический метод `make` создает новый экземпляр коллекции. См. раздел Создание коллекций.

`map()`

Метод `map` выполняет итерацию по коллекции и передает каждое значение указанному замыканию. Замыкание может изменить элемент и вернуть его, образуя новую коллекцию измененных элементов:

```
$collection = collect([1, 2, 3, 4, 5]);

$multiplied = $collection->map(function ($item, $key) {
    return $item * 2;
});

$multiplied->all();

// [2, 4, 6, 8, 10]
```

> {note} Как и большинство других методов коллекции, `map` возвращает новый экземпляр коллекции; он не модифицирует коллекцию. Если вы хотите преобразовать исходную коллекцию, используйте метод `transform`.

`mapInto()`

Метод `mapInto()` выполняет итерацию коллекции, создавая новый экземпляр указанного класса, и передавая значение в его конструктор:

```
class Currency
{
```

```
    /**
     * Создать новый экземпляр валюты.
     *
     * @param  string  $code
     * @return void
     */
    function __construct(string $code)
    {
        $this->code = $code;
    }
}

$collection = collect(['USD', 'EUR', 'GBP']);

$currencies = $collection->mapInto(Currency::class);

$currencies->all();

// [Currency('USD'), Currency('EUR'), Currency('GBP')]
```

mapSpread()

Метод `mapSpread` выполняет итерацию по элементам коллекции, передавая значение каждого вложенного элемента в указанное замыкание. Замыкание может изменить элемент и вернуть его, таким образом формируя новую коллекцию измененных элементов:

```
$collection = collect([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]);

$chunks = $collection->chunk(2);

$sequence = $chunks->mapSpread(function ($even, $odd) {
    return $even + $odd;
});

$sequence->all();

// [1, 5, 9, 13, 17]
```

mapToGroups()

Метод `mapToGroups` группирует элементы коллекции по указанному замыканию. Замыкание должно возвращать ассоциативный массив, содержащий одну пару ключ / значение, таким образом формируя новую коллекцию сгруппированных значений:

```
$collection = collect([
    [
        'name' => 'John Doe',
        'department' => 'Sales',
    ],
    [
```

```
        'name' => 'Jane Doe',
        'department' => 'Sales',
    ],
    [
        'name' => 'Johnny Doe',
        'department' => 'Marketing',
    ]
]);

$grouped = $collection->mapToGroups(function ($item, $key) {
    return [$item['department'] => $item['name']];
});

$grouped->all();

/*
    [
        'Sales' => ['John Doe', 'Jane Doe'],
        'Marketing' => ['Johnny Doe'],
    ]
*/

$grouped->get('Sales')->all();

// ['John Doe', 'Jane Doe']
```

mapWithKeys()

Метод `mapWithKeys` выполняет итерацию по коллекции и передает каждое значение в указанное замыкание. Замыкание должно возвращать ассоциативный массив, содержащий одну пару ключ / значение:

```
$collection = collect([
    [
        'name' => 'John',
        'department' => 'Sales',
        'email' => 'john@example.com',
    ],
    [
        'name' => 'Jane',
        'department' => 'Marketing',
        'email' => 'jane@example.com',
    ]
]);

$keyed = $collection->mapWithKeys(function ($item, $key) {
    return [$item['email'] => $item['name']];
});

$keyed->all();

/*
    [
```

```
        'john@example.com' => 'John',
        'jane@example.com' => 'Jane',
    ]
*/
```

max()

Метод `max` возвращает максимальное значение переданного ключа:

```
$max = collect([
    ['foo' => 10],
    ['foo' => 20]
])->max('foo');

// 20

$max = collect([1, 2, 3, 4, 5])->max();

// 5
```

median()

Метод `median` возвращает медиану переданного ключа:

```
$median = collect([
    ['foo' => 10],
    ['foo' => 10],
    ['foo' => 20],
    ['foo' => 40]
])->median('foo');

// 15

$median = collect([1, 1, 2, 4])->median();

// 1.5
```

merge()

Метод `merge` объединяет переданный массив или коллекцию с исходной коллекцией. Если строковый ключ в переданных элементах соответствует строковому ключу в исходной коллекции, значение переданного элемента перезапишет значение в исходной коллекции:

```
$collection = collect(['product_id' => 1, 'price' => 100]);

$merged = $collection->merge(['price' => 200, 'discount' => false]);

$merged->all();
```

```
// ['product_id' => 1, 'price' => 200, 'discount' => false]
```

Если ключи переданных элементов являются числовыми, значения будут добавлены в конец коллекции:

```
$collection = collect(['Desk', 'Chair']);

$merged = $collection->merge(['Bookcase', 'Door']);

$merged->all();

// ['Desk', 'Chair', 'Bookcase', 'Door']
```

mergeRecursive()

Метод `mergeRecursive` рекурсивно объединяет переданный массив или коллекцию с исходной коллекцией. Если строковый ключ в переданных элементах совпадает со строковым ключом в исходной коллекции, тогда значения этих ключей объединяются в массив, и это делается рекурсивно:

```
$collection = collect(['product_id' => 1, 'price' => 100]);

$merged = $collection->mergeRecursive([
    'product_id' => 2,
    'price' => 200,
    'discount' => false
]);

$merged->all();

// ['product_id' => [1, 2], 'price' => [100, 200], 'discount' => false]
```

min()

Метод `min` возвращает минимальное значение переданного ключа:

```
$min = collect([['foo' => 10], ['foo' => 20]])->min('foo');

// 10

$min = collect([1, 2, 3, 4, 5])->min();

// 1
```

mode()

Метод `mode` возвращает значение моды указанного ключа:

```
$mode = collect([
    ['foo' => 10],
    ['foo' => 10],
    ['foo' => 20],
    ['foo' => 40]
])->mode('foo');

// [10]

$mode = collect([1, 1, 2, 4])->mode();

// [1]

$mode = collect([1, 1, 2, 2])->mode();

// [1, 2]
```

nth()

Метод `nth` создает новую коллекцию, состоящую из каждого `n` -го элемента:

```
$collection = collect(['a', 'b', 'c', 'd', 'e', 'f']);

$collection->nth(4);

// ['a', 'e']
```

При желании вы можете передать начальное смещение в качестве второго аргумента:

```
$collection->nth(4, 1);

// ['b', 'f']
```

only()

Метод `only` возвращает элементы коллекции только с указанными ключами:

```
$collection = collect([
    'product_id' => 1,
    'name' => 'Desk',
    'price' => 100,
    'discount' => false
]);

$filtered = $collection->only(['product_id', 'name']);

$filtered->all();
```

```
// ['product_id' => 1, 'name' => 'Desk']
```

Противоположным методу `only` является метод except.

> {tip} Поведение этого метода изменяется при использовании коллекций Eloquent.

pad()

Метод `pad` дополнит коллекцию определенным значением, пока коллекция не достигнет указанного размера. Этот метод ведет себя как функция array_pad PHP.

Для дополнения слева следует указать отрицательный размер. Если абсолютное значение указанного размера меньше или равно длине массива, заполнение не произойдет:

```
$collection = collect(['A', 'B', 'C']);

$filtered = $collection->pad(5, 0);

$filtered->all();

// ['A', 'B', 'C', 0, 0]

$filtered = $collection->pad(-5, 0);

$filtered->all();

// [0, 0, 'A', 'B', 'C']
```

partition()

Метод `partition` может быть объединен с деструктуризацией массива PHP для разделения элементов, прошедших указанную проверку истинности, от тех, которые ее не прошли:

```
$collection = collect([1, 2, 3, 4, 5, 6]);

[$underThree, $equalOrAboveThree] = $collection->partition(function ($i) {
    return $i < 3;
});

$underThree->all();

// [1, 2]

$equalOrAboveThree->all();

// [3, 4, 5, 6]
```

pipe()

Метод `pipe` передает коллекцию указанному замыканию и возвращает результат выполненного замыкания:

```
$collection = collect([1, 2, 3]);

$piped = $collection->pipe(function ($collection) {
    return $collection->sum();
});

// 6
```

pipeInto()

Метод `pipeInto` создает новый экземпляр указанного класса и передает коллекцию в конструктор:

```
class ResourceCollection
{
    /**
     * Экземпляр Collection.
     */
    public $collection;

    /**
     * Создать новый экземпляр ResourceCollection.
     *
     * @param  Collection  $collection
     * @return void
     */
    public function __construct(Collection $collection)
    {
        $this->collection = $collection;
    }
}

$collection = collect([1, 2, 3]);

$resource = $collection->pipeInto(ResourceCollection::class);

$resource->collection->all();

// [1, 2, 3]
```

pipeThrough()

Метод `pipeThrough` передает коллекцию указанному массиву замыканий и возвращает результат выполненных замыканий:

```
$collection = collect([1, 2, 3]);
```

```
$result = $collection->pipeThrough([
    function ($collection) {
        return $collection->merge([4, 5]);
    },
    function ($collection) {
        return $collection->sum();
    },
]);

// 15
```

pluck()

Метод `pluck` извлекает все значения для указанного ключа:

```
$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'Desk'],
    ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$plucked = $collection->pluck('name');

$plucked->all();

// ['Desk', 'Chair']
```

Вы также можете задать ключ результирующей коллекции:

```
$plucked = $collection->pluck('name', 'product_id');

$plucked->all();

// ['prod-100' => 'Desk', 'prod-200' => 'Chair']
```

Метод `pluck` также поддерживает получение вложенных значений с использованием «точечной нотации»:

```
$collection = collect([
    [
        'name' => 'Laracon',
        'speakers' => [
            'first_day' => ['Rosa', 'Judith'],
        ],
    ],
    [
        'name' => 'VueConf',
        'speakers' => [
            'first_day' => ['Abigail', 'Joey'],
        ],
    ],
]);
```

```
$plucked = $collection->pluck('speakers.first_day');

$plucked->all();

// [['Rosa', 'Judith'], ['Abigail', 'Joey']]
```

Если существуют повторяющиеся ключи, последний соответствующий элемент будет вставлен в результирующую коллекцию:

```
$collection = collect([
    ['brand' => 'Tesla',  'color' => 'red'],
    ['brand' => 'Pagani', 'color' => 'white'],
    ['brand' => 'Tesla',  'color' => 'black'],
    ['brand' => 'Pagani', 'color' => 'orange'],
]);

$plucked = $collection->pluck('color', 'brand');

$plucked->all();

// ['Tesla' => 'black', 'Pagani' => 'orange']
```

pop()

Метод `pop` удаляет и возвращает последний элемент из коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->pop();

// 5

$collection->all();

// [1, 2, 3, 4]
```

Вы можете передать целое число методу `pop`, чтобы удалить (с возвратом) несколько элементов из конца коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->pop(3);

// collect([5, 4, 3])

$collection->all();

// [1, 2]
```

prepend()

Метод `prepend` добавляет элемент в начало коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->prepend(0);

$collection->all();

// [0, 1, 2, 3, 4, 5]
```

Вы также можете передать второй аргумент, чтобы указать ключ добавляемого элемента:

```
$collection = collect(['one' => 1, 'two' => 2]);

$collection->prepend(0, 'zero');

$collection->all();

// ['zero' => 0, 'one' => 1, 'two' => 2]
```

pull()

Метод `pull` удаляет и возвращает элемент из коллекции по его ключу:

```
$collection = collect(['product_id' => 'prod-100', 'name' => 'Desk']);

$collection->pull('name');

// 'Desk'

$collection->all();

// ['product_id' => 'prod-100']
```

push()

Метод `push` добавляет элемент в конец коллекции:

```
$collection = collect([1, 2, 3, 4]);

$collection->push(5);

$collection->all();

// [1, 2, 3, 4, 5]
```

put()

Метод `put` помещает указанные ключ и значение в коллекцию:

```
$collection = collect(['product_id' => 1, 'name' => 'Desk']);

$collection->put('price', 100);

$collection->all();

// ['product_id' => 1, 'name' => 'Desk', 'price' => 100]
```

random()

Метод `random` возвращает случайный элемент из коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->random();

// 4 - (retrieved randomly)
```

Вы можете передать целое число в `random`, чтобы указать, сколько случайных элементов вы хотите получить. Коллекция элементов всегда возвращается при явной передаче количества элементов, которые вы хотите получить:

```
$random = $collection->random(3);

$random->all();

// [2, 4, 5] - (retrieved randomly)
```

Если в экземпляре коллекции меньше элементов, чем запрошено, метод `random` сгенерирует исключение `InvalidArgumentException`.

range()

Метод `range` возвращает коллекцию, содержащую целые числа в указанном диапазоне:

```
$collection = collect()->range(3, 6);

$collection->all();

// [3, 4, 5, 6]
```

reduce()

Метод `reduce` сокращает коллекцию до одного значения, передавая результат каждой итерации следующей итерации:

```
$collection = collect([1, 2, 3]);

$total = $collection->reduce(function ($carry, $item) {
    return $carry + $item;
});

// 6
```

Значение `$carry` первой итерации равно `null` ; однако вы можете указать его начальное значение, передав второй аргумент методу `reduce` :

```
$collection->reduce(function ($carry, $item) {
    return $carry + $item;
}, 4);

// 10
```

Метод `reduce` также передает ключи массива ассоциативных коллекций указанному замыканию:

```
$collection = collect([
    'usd' => 1400,
    'gbp' => 1200,
    'eur' => 1000,
]);

$ratio = [
    'usd' => 1,
    'gbp' => 1.37,
    'eur' => 1.22,
];

$collection->reduce(function ($carry, $value, $key) use ($ratio) {
    return $carry + ($value * $ratio[$key]);
});

// 4264
```

`reduceSpread()`

Метод `reduceSpread` сокращает коллекцию до массива значений, передавая результаты каждой итерации в следующую итерацию. Этот метод похож на метод `reduce` ; однако он может принимать несколько начальных значений:

```
[$creditsRemaining, $batch] = Image::where('status', 'unprocessed')
    ->get()
```

```
    ->reduceSpread(function ($creditsRemaining, $batch, $image) {
        if ($creditsRemaining >= $image->creditsRequired()) {
            $batch->push($image);

            $creditsRemaining -= $image->creditsRequired();
        }

        return [$creditsRemaining, $batch];
    }, $creditsAvailable, collect());
```

reject()

Метод `reject` фильтрует коллекцию, используя переданное замыкание. Замыкание должно возвращать `true`, если элемент должен быть удален из результирующей коллекции:

```
$collection = collect([1, 2, 3, 4]);

$filtered = $collection->reject(function ($value, $key) {
    return $value > 2;
});

$filtered->all();

// [1, 2]
```

Противоположным методу `reject` является метод `filter`.

replace()

Метод `replace` ведет себя аналогично методу `merge`; однако, помимо перезаписи совпадающих элементов, имеющих строковые ключи, метод `replace` также перезаписывает элементы в коллекции, у которых есть совпадающие числовые ключи:

```
$collection = collect(['Taylor', 'Abigail', 'James']);

$replaced = $collection->replace([1 => 'Victoria', 3 => 'Finn']);

$replaced->all();

// ['Taylor', 'Victoria', 'James', 'Finn']
```

replaceRecursive()

Этот метод работает как и `replace`, но он будет повторяться в массивах и применять тот же процесс замены к внутренним значениям:

```
$collection = collect([
    'Taylor',
    'Abigail',
```

```
    [
        'James',
        'Victoria',
        'Finn'
    ]
]);

$replaced = $collection->replaceRecursive([
    'Charlie',
    2 => [1 => 'King']
]);

$replaced->all();

// ['Charlie', 'Abigail', ['James', 'King', 'Finn']]
```

reverse()

Метод `reverse` меняет порядок элементов коллекции на обратный, сохраняя исходные ключи:

```
$collection = collect(['a', 'b', 'c', 'd', 'e']);

$reversed = $collection->reverse();

$reversed->all();

/*
    [
        4 => 'e',
        3 => 'd',
        2 => 'c',
        1 => 'b',
        0 => 'a',
    ]
*/
```

search()

Метод `search` ищет в коллекции указанное значение и возвращает его ключ, если он найден. Если элемент не найден, возвращается `false` :

```
$collection = collect([2, 4, 6, 8]);

$collection->search(4);

// 1
```

Поиск выполняется с использованием «гибкого» сравнения, то есть строка с целым значением будет считаться равной целому числу того же значения. Чтобы использовать «жесткое» сравнение, передайте `true` в качестве второго аргумента метода:

```
collect([2, 4, 6, 8])->search('4', $strict = true);

// false
```

В качестве альтернативы вы можете передать собственное замыкание для поиска первого элемента, который проходит указанный тест на истинность:

```
collect([2, 4, 6, 8])->search(function ($item, $key) {
    return $item > 5;
});

// 2
```

shift()

Метод `shift` удаляет и возвращает первый элемент из коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->shift();

// 1

$collection->all();

// [2, 3, 4, 5]
```

Вы можете передать целое число методу `shift` , чтобы удалить (с возвратом) несколько элементов из начала коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->shift(3);

// collect([1, 2, 3])

$collection->all();

// [4, 5]
```

shuffle()

Метод `shuffle` случайным образом перемешивает элементы в коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);

$shuffled = $collection->shuffle();

$shuffled->all();

// [3, 2, 5, 1, 4] - (generated randomly)
```

sliding()

Метод `sliding` возвращает новую фрагментированную коллекцию, представляющих вид «скользящего окна» элементов:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunks = $collection->sliding(2);

$chunks->toArray();

// [[1, 2], [2, 3], [3, 4], [4, 5]]
```

Это особенно полезно в сочетании с методом `eachSpread` :

```
$transactions->sliding(2)->eachSpread(function ($previous, $current) {
    $current->total = $previous->total + $current->amount;
});
```

При желании вы можете передать второе значение `step` , которое определяет расстояние между первым элементом каждого фрагмента:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunks = $collection->sliding(3, step: 2);

$chunks->toArray();

// [[1, 2, 3], [3, 4, 5]]
```

skip()

Метод `skip` возвращает новую коллекцию с указанным количеством удаляемых из начала коллекции элементов:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

$collection = $collection->skip(4);
```

```
$collection->all();

// [5, 6, 7, 8, 9, 10]
```

skipUntil()

Метод `skipUntil` пропускает элементы из коллекции до тех пор, пока переданное замыкание не вернет `true`, а затем вернет оставшиеся элементы в коллекции как новый экземпляр коллекции:

```
$collection = collect([1, 2, 3, 4]);

$subset = $collection->skipUntil(function ($item) {
    return $item >= 3;
});

$subset->all();

// [3, 4]
```

Вы также можете передать простое значение методу `skipUntil`, чтобы пропустить все элементы, пока не будет найдено указанное значение:

```
$collection = collect([1, 2, 3, 4]);

$subset = $collection->skipUntil(3);

$subset->all();

// [3, 4]
```

> {note} Если указанное значение не найдено или замыкание никогда не возвращает `true`, то метод `skipUntil` вернет пустую коллекцию.

skipWhile()

Метод `skipWhile` пропускает элементы из коллекции, пока указанное замыкание возвращает `true`, а затем возвращает оставшиеся элементы в коллекции как новую коллекцию:

```
$collection = collect([1, 2, 3, 4]);

$subset = $collection->skipWhile(function ($item) {
    return $item <= 3;
});

$subset->all();

// [4]
```

> {note} Если замыкание никогда не возвращает `false`, то метод `skipWhile` вернет пустую коллекцию.

slice()

Метод `slice` возвращает фрагмент коллекции, начиная с указанного индекса:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

$slice = $collection->slice(4);

$slice->all();

// [5, 6, 7, 8, 9, 10]
```

Если вы хотите ограничить размер возвращаемого фрагмента, то передайте желаемый размер в качестве второго аргумента метода:

```
$slice = $collection->slice(4, 2);

$slice->all();

// [5, 6]
```

Возвращенный фрагмент по умолчанию сохранит ключи. Если вы не хотите сохранять исходные ключи, вы можете использовать метод `values`, чтобы переиндексировать их.

sole()

Метод `sole` возвращает первый элемент, который является единственным элементом в коллекции, прошедшим указанный тест истинности:

```
collect([1, 2, 3, 4])->sole(function ($value, $key) {
    return $value === 2;
});

// 2
```

Вы также можете передать пару ключ / значение методу `sole`, который вернет первый и единственный элемент коллекции, соответствующий переданной паре:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
]);

$collection->sole('product', 'Chair');
```

```
// ['product' => 'Chair', 'price' => 100]
```

В качестве альтернативы вы также можете вызвать метод `sole` без аргументов, чтобы получить первый элемент, который является единственным элементом в коллекции:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
]);

$collection->sole();

// ['product' => 'Desk', 'price' => 200]
```

Если в коллекции нет элементов, которые должны быть возвращены методом `sole`, то будет выброшено исключение `\Illuminate\Collections\ItemNotFoundException`. Если таких элементов в коллекции более одного элемента, то будет выброшено исключение `\Illuminate\Collections\MultipleItemsFoundException`.

some()

Псевдоним для метода `contains`.

sort()

Метод `sort` сортирует коллекцию. В отсортированной коллекции хранятся исходные ключи массива, поэтому в следующем примере мы будем использовать метод `values` для сброса ключей для последовательной нумерации индексов:

```
$collection = collect([5, 3, 1, 2, 4]);

$sorted = $collection->sort();

$sorted->values()->all();

// [1, 2, 3, 4, 5]
```

Если ваши потребности в сортировке более сложны, вы можете передать замыкание методу `sort` с вашим собственным алгоритмом. Обратитесь к документации PHP по `uasort`, который используется внутри метода `sort`.

> {tip} Если вам нужно отсортировать коллекцию вложенных массивов или объектов, то см. методы `sortBy` и `sortByDesc`.

sortBy()

Метод `sortBy` сортирует коллекцию по указанному ключу. В отсортированной коллекции хранятся исходные ключи массива, поэтому в следующем примере мы будем использовать метод `values` для сброса ключей для последовательной нумерации индексов:

```
$collection = collect([
    ['name' => 'Desk', 'price' => 200],
    ['name' => 'Chair', 'price' => 100],
    ['name' => 'Bookcase', 'price' => 150],
]);

$sorted = $collection->sortBy('price');

$sorted->values()->all();

/*
    [
        ['name' => 'Chair', 'price' => 100],
        ['name' => 'Bookcase', 'price' => 150],
        ['name' => 'Desk', 'price' => 200],
    ]
*/
```

Метод `sortBy` принимает флаги типа сортировки в качестве второго аргумента:

```
$collection = collect([
    ['title' => 'Item 1'],
    ['title' => 'Item 12'],
    ['title' => 'Item 3'],
]);

$sorted = $collection->sortBy('title', SORT_NATURAL);

$sorted->values()->all();

/*
    [
        ['title' => 'Item 1'],
        ['title' => 'Item 3'],
        ['title' => 'Item 12'],
    ]
*/
```

В качестве альтернативы вы можете передать собственное замыкание, чтобы определить, как сортировать значения коллекции:

```
$collection = collect([
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]);

$sorted = $collection->sortBy(function ($product, $key) {
    return count($product['colors']);
});

$sorted->values()->all();
```

```
/*
    [
        ['name' => 'Chair', 'colors' => ['Black']],
        ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
        ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
    ]
*/
```

Если вы хотите отсортировать свою коллекцию по нескольким атрибутам, вы можете передать массив операций сортировки методу `sortBy` . Каждая операция сортировки должна быть массивом, состоящим из атрибута, по которому вы хотите сортировать, и направления желаемой сортировки:

```
$collection = collect([
    ['name' => 'Taylor Otwell', 'age' => 34],
    ['name' => 'Abigail Otwell', 'age' => 30],
    ['name' => 'Taylor Otwell', 'age' => 36],
    ['name' => 'Abigail Otwell', 'age' => 32],
]);

$sorted = $collection->sortBy([
    ['name', 'asc'],
    ['age', 'desc'],
]);

$sorted->values()->all();

/*
    [
        ['name' => 'Abigail Otwell', 'age' => 32],
        ['name' => 'Abigail Otwell', 'age' => 30],
        ['name' => 'Taylor Otwell', 'age' => 36],
        ['name' => 'Taylor Otwell', 'age' => 34],
    ]
*/
```

При сортировке коллекции по нескольким атрибутам вы также можете указать замыкания, определяющие каждую операцию сортировки:

```
$collection = collect([
    ['name' => 'Taylor Otwell', 'age' => 34],
    ['name' => 'Abigail Otwell', 'age' => 30],
    ['name' => 'Taylor Otwell', 'age' => 36],
    ['name' => 'Abigail Otwell', 'age' => 32],
]);

$sorted = $collection->sortBy([
    fn ($a, $b) => $a['name'] <=> $b['name'],
    fn ($a, $b) => $b['age'] <=> $a['age'],
]);

$sorted->values()->all();
```

```
/*
    [
        ['name' => 'Abigail Otwell', 'age' => 32],
        ['name' => 'Abigail Otwell', 'age' => 30],
        ['name' => 'Taylor Otwell', 'age' => 36],
        ['name' => 'Taylor Otwell', 'age' => 34],
    ]
*/
```

sortByDesc()

Этот метод имеет ту же сигнатуру, что и метод `sortBy`, но отсортирует коллекцию в обратном порядке.

sortDesc()

Этот метод сортирует коллекцию в порядке, обратном методу `sort`:

```
$collection = collect([5, 3, 1, 2, 4]);

$sorted = $collection->sortDesc();

$sorted->values()->all();

// [5, 4, 3, 2, 1]
```

В отличие от `sort`, вы не можете передавать замыкание в `sortDesc`. Вместо этого вы должны использовать метод `sort` и инвертировать ваше сравнение.

sortKeys()

Метод `sortKeys` сортирует коллекцию по ключам базового ассоциативного массива:

```
$collection = collect([
    'id' => 22345,
    'first' => 'John',
    'last' => 'Doe',
]);

$sorted = $collection->sortKeys();

$sorted->all();

/*
    [
        'first' => 'John',
        'id' => 22345,
        'last' => 'Doe',
    ]
*/
```

sortKeysDesc()

Этот метод имеет ту же сигнатуру, что и метод `sortKeys`, но отсортирует коллекцию в обратном порядке.

sortKeysUsing()

Метод `sortKeysUsing` сортирует коллекцию по ключам ассоциативного массива с помощью замыкания:

```
$collection = collect([
    'ID' => 22345,
    'first' => 'John',
    'last' => 'Doe',
]);

$sorted = $collection->sortKeysUsing('strnatcasecmp');

$sorted->all();

/*
    [
        'first' => 'John',
        'ID' => 22345,
        'last' => 'Doe',
    ]
*/
```

Замыкание должно быть функцией сравнения, возвращающей целое число, которое меньше, равно или больше нуля. Для получения дополнительной информации обратитесь к документации по `uksort`, которая является функцией PHP, используемой методом `sortKeysUsing`.

splice()

Метод `splice` удаляет и возвращает фрагмент элементов, начиная с указанного индекса:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2);

$chunk->all();

// [3, 4, 5]

$collection->all();

// [1, 2]
```

Вы можете передать второй аргумент, чтобы ограничить размер результирующей коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2, 1);

$chunk->all();

// [3]

$collection->all();

// [1, 2, 4, 5]
```

Кроме того, вы можете передать третий аргумент, содержащий новые элементы, чтобы заменить элементы, удаленные из коллекции:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2, 1, [10, 11]);

$chunk->all();

// [3]

$collection->all();

// [1, 2, 10, 11, 4, 5]
```

split()

Метод `split` разбивает коллекцию на указанное количество групп:

```
$collection = collect([1, 2, 3, 4, 5]);

$groups = $collection->split(3);

$groups->all();

// [[1, 2], [3, 4], [5]]
```

splitIn()

Метод `splitIn` разбивает коллекцию на указанное количество групп, полностью заполняя нетерминальные группы перед тем, как выделить остаток последней группе:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

$groups = $collection->splitIn(3);

$groups->all();

// [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10]]
```

sum()

Метод `sum` возвращает сумму всех элементов в коллекции:

```
collect([1, 2, 3, 4, 5])->sum();

// 15
```

Если коллекция содержит вложенные массивы или объекты, вы должны передать ключ, который будет использоваться для определения суммирования значений:

```
$collection = collect([
    ['name' => 'JavaScript: The Good Parts', 'pages' => 176],
    ['name' => 'JavaScript: The Definitive Guide', 'pages' => 1096],
]);

$collection->sum('pages');

// 1272
```

Кроме того, вы можете передать собственное замыкание, чтобы определить, какие значения коллекции суммировать:

```
$collection = collect([
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]);

$collection->sum(function ($product) {
    return count($product['colors']);
});

// 6
```

take()

Метод `take` возвращает новую коллекцию с указанным количеством элементов:

```
$collection = collect([0, 1, 2, 3, 4, 5]);

$chunk = $collection->take(3);

$chunk->all();

// [0, 1, 2]
```

Вы также можете передать отрицательное целое число, чтобы получить указанное количество элементов из конца коллекции:

```
$collection = collect([0, 1, 2, 3, 4, 5]);

$chunk = $collection->take(-2);

$chunk->all();

// [4, 5]
```

`takeUntil()`

Метод `takeUntil` возвращает элементы коллекции, пока указанное замыкание не вернет `true`:

```
$collection = collect([1, 2, 3, 4]);

$subset = $collection->takeUntil(function ($item) {
    return $item >= 3;
});

$subset->all();

// [1, 2]
```

Вы также можете передать простое значение методу `takeUntil`, чтобы получать элементы, пока не будет найдено указанное значение:

```
$collection = collect([1, 2, 3, 4]);

$subset = $collection->takeUntil(3);

$subset->all();

// [1, 2]
```

> {note} Если указанное значение не найдено или замыкание никогда не возвращает `true`, то метод `takeUntil` вернет все элементы коллекции.

takeWhile()

Метод `takeWhile` возвращает элементы коллекции до тех пор, пока указанное замыкание не вернет `false`:

```
$collection = collect([1, 2, 3, 4]);

$subset = $collection->takeWhile(function ($item) {
    return $item < 3;
});

$subset->all();

// [1, 2]
```

> {note} Если замыкание никогда не возвращает `false`, метод `takeWhile` вернет все элементы коллекции.

tap()

Метод `tap` передает коллекцию указанному замыканию, позволяя вам «перехватить» коллекцию в определенный момент и сделать что-то с элементами, не затрагивая саму коллекцию. Затем коллекция возвращается методом `tap`:

```
collect([2, 4, 3, 1, 5])
    ->sort()
    ->tap(function ($collection) {
        Log::debug('Values after sorting', $collection->values()->all());
    })
    ->shift();

// 1
```

times()

Статический метод `times` создает новую коллекцию, вызывая переданное замыкание указанное количество раз:

```
$collection = Collection::times(10, function ($number) {
    return $number * 9;
});

$collection->all();

// [9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
```

toArray()

Метод `toArray` преобразует коллекцию в простой массив PHP. Если значениями коллекции являются модели Eloquent, то модели также будут преобразованы в массивы:

```
$collection = collect(['name' => 'Desk', 'price' => 200]);

$collection->toArray();

/*
    [
        ['name' => 'Desk', 'price' => 200],
    ]
*/
```

> {note} Метод `toArray` также преобразует все вложенные объекты коллекции, которые являются экземпляром `Arrayable`, в массив. Если вы хотите получить необработанный массив, лежащий в основе коллекции, используйте вместо этого метод `all`.

toJson()

Метод `toJson` преобразует коллекцию в сериализованную строку JSON:

```
$collection = collect(['name' => 'Desk', 'price' => 200]);

$collection->toJson();

// '{"name":"Desk", "price":200}'
```

transform()

Метод `transform` выполняет итерацию коллекции и вызывает указанное замыкание для каждого элемента в коллекции. Элементы в коллекции будут заменены значениями, возвращаемыми замыканием:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->transform(function ($item, $key) {
    return $item * 2;
});

$collection->all();

// [2, 4, 6, 8, 10]
```

> {note} В отличие от большинства других методов коллекции, `transform` модифицирует коллекцию. Если вы хотите вместо этого создать новую коллекцию, используйте метод `map`.

undot()

Метод `undot` расширяет одноуровневую коллекцию, в которой используется «точечная нотация», в многомерную коллекцию:

```
$person = collect([
    'name.first_name' => 'Marie',
    'name.last_name' => 'Valentine',
    'address.line_1' => '2992 Eagle Drive',
    'address.line_2' => '',
    'address.suburb' => 'Detroit',
    'address.state' => 'MI',
    'address.postcode' => '48219'
])

$person = $person->undot();

$person->toArray();

/*
    [
        "name" => [
            "first_name" => "Marie",
            "last_name" => "Valentine",
        ],
        "address" => [
            "line_1" => "2992 Eagle Drive",
            "line_2" => "",
            "suburb" => "Detroit",
            "state" => "MI",
            "postcode" => "48219",
        ],
    ]
*/
```

union()

Метод `union` добавляет переданный массив в коллекцию. Если переданный массив содержит ключи, которые уже находятся в исходной коллекции, предпочтительнее будут значения исходной коллекции:

```
$collection = collect([1 => ['a'], 2 => ['b']]);

$union = $collection->union([3 => ['c'], 1 => ['d']]);

$union->all();

// [1 => ['a'], 2 => ['b'], 3 => ['c']]
```

unique()

Метод `unique` возвращает все уникальные элементы коллекции. Возвращенная коллекция сохраняет исходные ключи массива, поэтому в следующем примере мы будем использовать метод `values` для сброса ключей для последовательной нумерации индексов:

```
$collection = collect([1, 1, 2, 2, 3, 4, 2]);

$unique = $collection->unique();

$unique->values()->all();

// [1, 2, 3, 4]
```

При работе с вложенными массивами или объектами вы можете указать ключ, используемый для определения уникальности:

```
$collection = collect([
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'iPhone 5', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
    ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],
]);

$unique = $collection->unique('brand');

$unique->values()->all();

/*
    [
        ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
        ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
    ]
*/
```

Наконец, вы также можете передать собственное замыкание методу `unique`, чтобы указать, какое значение должно определять уникальность элемента:

```
$unique = $collection->unique(function ($item) {
    return $item['brand'].$item['type'];
});

$unique->values()->all();

/*
    [
        ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
        ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],
        ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
        ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],
    ]
*/
```

Метод `unique` использует «гибкое» сравнение при проверке значений элементов, то есть строка с целым значением будет считаться равной целому числу того же значения. Используйте метод `uniqueStrict` для фильтрации с использованием «жесткого» сравнения.

> {tip} Поведение этого метода изменяется при использовании коллекций Eloquent.

uniqueStrict()

Этот метод имеет ту же сигнатуру, что и метод `unique` ; однако, все значения сравниваются с использованием «жесткого» сравнения.

unless()

Метод `unless` выполнит указанное замыкание, если первый аргумент, переданный методу, оценивается как `false` :

```
$collection = collect([1, 2, 3]);

$collection->unless(true, function ($collection) {
    return $collection->push(4);
});

$collection->unless(false, function ($collection) {
    return $collection->push(5);
});

$collection->all();

// [1, 2, 3, 5]
```

Второе замыкание, переданное методу `unless` , будет выполнено, если первый аргумент, переданный методу, оценивается как `true` :

```
$collection = collect([1, 2, 3]);

$collection->unless(true, function ($collection) {
    return $collection->push(4);
}, function ($collection) {
    return $collection->push(5);
});

$collection->all();

// [1, 2, 3, 5]
```

Противоположным методу `unless` является метод `when` .

unlessEmpty()

Псевдоним для метода `whenNotEmpty` .

unlessNotEmpty()

Псевдоним для метода `whenEmpty`.

unwrap()

Статический метод `unwrap` возвращает базовые элементы коллекции из указанного значения, когда это применимо:

```
Collection::unwrap(collect('John Doe'));

// ['John Doe']

Collection::unwrap(['John Doe']);

// ['John Doe']

Collection::unwrap('John Doe');

// 'John Doe'
```

value()

Метод `value` извлекает указанное значение первого элемента коллекции:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Speaker', 'price' => 400],
]);

$value = $collection->value('price');

// 200
```

values()

Метод `values` возвращает новую коллекцию с ключами, сброшенными на последовательные целые числа:

```
$collection = collect([
    10 => ['product' => 'Desk', 'price' => 200],
    11 => ['product' => 'Desk', 'price' => 200],
]);

$values = $collection->values();

$values->all();

/*
    [
```

```
        0 => ['product' => 'Desk', 'price' => 200],
        1 => ['product' => 'Desk', 'price' => 200],
    ]
*/
```

when()

Метод `when` выполнит указанное замыкание, если первый аргумент, переданный методу, оценивается как `true`. Экземпляр коллекции и первый аргумент, переданный методу `when`, будут переданы замыканию:

```
$collection = collect([1, 2, 3]);

$collection->when(true, function ($collection, $value) {
    return $collection->push(4);
});

$collection->when(false, function ($collection, $value) {
    return $collection->push(5);
});

$collection->all();

// [1, 2, 3, 4]
```

Второе замыкание, переданное методу `when`, будет выполнено, если первый аргумент, переданный методу, оценивается как `false`:

```
$collection = collect([1, 2, 3]);

$collection->when(false, function ($collection, $value) {
    return $collection->push(4);
}, function ($collection) {
    return $collection->push(5);
});

$collection->all();

// [1, 2, 3, 5]
```

Противоположным методу `when` является метод `unless`.

whenEmpty()

Метод `whenEmpty` выполнит указанное замыкание, если коллекция пуста:

```
$collection = collect(['Michael', 'Tom']);

$collection->whenEmpty(function ($collection) {
    return $collection->push('Adam');
```

```
});

$collection->all();

// ['Michael', 'Tom']


$collection = collect();

$collection->whenEmpty(function ($collection) {
    return $collection->push('Adam');
});

$collection->all();

// ['Adam']
```

Второе замыкание, переданное методу `whenEmpty`, будет выполнено, если коллекция не пуста:

```
$collection = collect(['Michael', 'Tom']);

$collection->whenEmpty(function ($collection) {
    return $collection->push('Adam');
}, function ($collection) {
    return $collection->push('Taylor');
});

$collection->all();

// ['Michael', 'Tom', 'Taylor']
```

Противоположным методу `whenEmpty` является метод `whenNotEmpty`.

whenNotEmpty()

Метод `whenNotEmpty` выполнит указанное замыкание, если коллекция не пуста:

```
$collection = collect(['michael', 'tom']);

$collection->whenNotEmpty(function ($collection) {
    return $collection->push('adam');
});

$collection->all();

// ['michael', 'tom', 'adam']


$collection = collect();

$collection->whenNotEmpty(function ($collection) {
    return $collection->push('adam');
```

```
});

$collection->all();

// []
```

Второе замыкание, переданное методу `whenNotEmpty` , будет выполнено, если коллекция пуста:

```
$collection = collect();

$collection->whenNotEmpty(function ($collection) {
    return $collection->push('adam');
}, function ($collection) {
    return $collection->push('taylor');
});

$collection->all();

// ['taylor']
```

Противоположным методу `whenNotEmpty` является метод `whenEmpty` .

where()

Метод `where` фильтрует коллекцию по указанной паре ключ / значение:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->where('price', 100);

$filtered->all();

/*
    [
        ['product' => 'Chair', 'price' => 100],
        ['product' => 'Door', 'price' => 100],
    ]
*/
```

Метод `where` использует «гибкое» сравнение при проверке значений элементов, что означает, что строка с целым значением будет считаться равной целому числу того же значения. Используйте метод `whereStrict` для фильтрации с использованием «жесткого» сравнения.

При желании вы можете передать оператор сравнения в качестве второго параметра.

```
$collection = collect([
    ['name' => 'Jim', 'deleted_at' => '2019-01-01 00:00:00'],
    ['name' => 'Sally', 'deleted_at' => '2019-01-02 00:00:00'],
    ['name' => 'Sue', 'deleted_at' => null],
]);

$filtered = $collection->where('deleted_at', '!=', null);

$filtered->all();

/*
    [
        ['name' => 'Jim', 'deleted_at' => '2019-01-01 00:00:00'],
        ['name' => 'Sally', 'deleted_at' => '2019-01-02 00:00:00'],
    ]
*/
```

whereStrict()

Этот метод имеет ту же сигнатуру, что и метод `where`; однако, все значения сравниваются с использованием «жесткого» сравнения.

whereBetween()

Метод `whereBetween` фильтрует коллекцию, определяя, находится ли переданное значение элемента в указанном диапазоне:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 80],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Pencil', 'price' => 30],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereBetween('price', [100, 200]);

$filtered->all();

/*
    [
        ['product' => 'Desk', 'price' => 200],
        ['product' => 'Bookcase', 'price' => 150],
        ['product' => 'Door', 'price' => 100],
    ]
*/
```

whereIn()

Метод `whereIn` удаляет элементы из коллекции, у которых значения отсутствуют в указанном массиве:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereIn('price', [150, 200]);

$filtered->all();

/*
    [
        ['product' => 'Desk', 'price' => 200],
        ['product' => 'Bookcase', 'price' => 150],
    ]
*/
```

Метод `whereIn` использует «гибкое» сравнение при проверке значений элементов, что означает, что строка с целым значением будет считаться равной целому числу того же значения. Используйте метод `whereInStrict` для фильтрации с использованием «жесткого» сравнения.

whereInStrict()

Этот метод имеет ту же сигнатуру, что и метод `whereIn`; однако, все значения сравниваются с использованием «жесткого» сравнения.

whereInstanceOf()

Метод `whereInstanceOf` фильтрует коллекцию по указанному типу класса:

```
use App\Models\User;
use App\Models\Post;

$collection = collect([
    new User,
    new User,
    new Post,
]);

$filtered = $collection->whereInstanceOf(User::class);

$filtered->all();

// [App\Models\User, App\Models\User]
```

whereNotBetween()

Метод `whereNotBetween` фильтрует коллекцию, определяя, находится ли переданное значение элемента вне указанного диапазона:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 80],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Pencil', 'price' => 30],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereNotBetween('price', [100, 200]);

$filtered->all();

/*
    [
        ['product' => 'Chair', 'price' => 80],
        ['product' => 'Pencil', 'price' => 30],
    ]
*/
```

whereNotIn()

Метод `whereNotIn` удаляет элементы из коллекции, если их значения присутствуют в указанном массиве:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereNotIn('price', [150, 200]);

$filtered->all();

/*
    [
        ['product' => 'Chair', 'price' => 100],
        ['product' => 'Door', 'price' => 100],
    ]
*/
```

Метод `whereNotIn` использует «гибкое» сравнение при проверке значений элементов, что означает, что строка с целым значением будет считаться равной целому числу того же значения. Используйте метод `whereNotInStrict` для фильтрации с использованием «жесткого» сравнения.

whereNotInStrict()

Этот метод имеет ту же сигнатуру, что и метод `whereNotIn` ; однако, все значения сравниваются с использованием «жесткого» сравнения.

whereNotNull()

Метод `whereNotNull` возвращает элементы из коллекции, для которых значение указанного ключа не равно `null` :

```
$collection = collect([
    ['name' => 'Desk'],
    ['name' => null],
    ['name' => 'Bookcase'],
]);

$filtered = $collection->whereNotNull('name');

$filtered->all();

/*
    [
        ['name' => 'Desk'],
        ['name' => 'Bookcase'],
    ]
*/
```

whereNull()

Метод `whereNull` возвращает элементы из коллекции, для которых значение указанного ключа равно `null` :

```
$collection = collect([
    ['name' => 'Desk'],
    ['name' => null],
    ['name' => 'Bookcase'],
]);

$filtered = $collection->whereNull('name');

$filtered->all();

/*
    [
        ['name' => null],
    ]
*/
```

wrap()

Статический метод `wrap` оборачивает указанное значение в коллекцию, если это применимо:

```
use Illuminate\Support\Collection;

$collection = Collection::wrap('John Doe');

$collection->all();

// ['John Doe']

$collection = Collection::wrap(['John Doe']);

$collection->all();

// ['John Doe']

$collection = Collection::wrap(collect('John Doe'));

$collection->all();

// ['John Doe']
```

zip()

Метод `zip` объединяет значения переданного массива со значениями исходной коллекции по их соответствующему индексу:

```
$collection = collect(['Chair', 'Desk']);

$zipped = $collection->zip([100, 200]);

$zipped->all();

// [['Chair', 100], ['Desk', 200]]
```

Сообщения высшего порядка

Коллекции также обеспечивают поддержку «сообщений высшего порядка», которые являются сокращениями для выполнения общих действий с коллекциями. Методы коллекции, которые предоставляют сообщения высшего порядка: `average`, `avg`, `contains`, `each`, `every`, `filter`, `first`, `flatMap`, `groupBy`, `keyBy`, `map`, `max`, `min`, `partition`, `reject`, `skipUntil`, `skipWhile`, `some`, `sortBy`, `sortByDesc`, `sum`, `takeUntil`, `takeWhile`, и `unique`.

К каждому сообщению высшего порядка можно получить доступ как к динамическому свойству экземпляра коллекции. Например, давайте использовать сообщение высшего порядка `each`, вызывая метод для каждого объекта коллекции:

```
use App\Models\User;

$users = User::where('votes', '>', 500)->get();

$users->each->markAsVip();
```

Точно так же мы можем использовать сообщение высшего порядка `sum`, чтобы собрать общее количество «голосов» для коллекции пользователей:

```
$users = User::where('group', 'Development')->get();

return $users->sum->votes;
```

Отложенные коллекции

Введение в отложенные коллекции

> {note} Прежде чем узнать больше об отложенных коллекциях Laravel, потратьте некоторое время на то, чтобы ознакомиться с генераторами PHP.

В дополнении к мощному классу `Collection`, класс `LazyCollection` использует генераторы PHP, чтобы вы могли работать с очень большим наборы данных при низком потреблении памяти.

Например, представьте, что ваше приложение должно обрабатывать файл журнала размером в несколько гигабайт, используя при этом методы коллекций Laravel для анализа журналов. Вместо одновременного чтения всего файла в память можно использовать отложенные коллекции, чтобы сохранить в памяти только небольшую часть файла в текущий момент:

```
use App\Models\LogEntry;
use Illuminate\Support\LazyCollection;

LazyCollection::make(function () {
    $handle = fopen('log.txt', 'r');

    while (($line = fgets($handle)) !== false) {
        yield $line;
    }
})->chunk(4)->map(function ($lines) {
    return LogEntry::fromLines($lines);
})->each(function (LogEntry $logEntry) {
    // Process the log entry...
});
```

Или представьте, что вам нужно перебрать 10 000 моделей Eloquent. При использовании традиционных коллекций Laravel все 10 000 моделей Eloquent должны быть загружены в память одновременно:

```
use App\Models\User;

$users = User::all()->filter(function ($user) {
    return $user->id > 500;
});
```

Однако, метод `cursor` построителя запросов возвращает экземпляр `LazyCollection`. Это позволяет вам по-прежнему выполнять только один запрос к базе данных, но при этом одновременно загружать в память только одну модель Eloquent. В этом примере замыкание метода `filter` не выполнится до тех пор, пока мы на самом деле не переберем каждого пользователя индивидуально, что позволяет значительно сократить использование памяти:

```
use App\Models\User;

$users = User::cursor()->filter(function ($user) {
    return $user->id > 500;
});

foreach ($users as $user) {
    echo $user->id;
}
```

Создание отложенных коллекций

Чтобы создать экземпляр отложенной коллекции, вы должны передать функцию генератора PHP методу `make` коллекции:

```
use Illuminate\Support\LazyCollection;

LazyCollection::make(function () {
    $handle = fopen('log.txt', 'r');

    while (($line = fgets($handle)) !== false) {
        yield $line;
    }
});
```

Контракт `Enumerable`

Почти все методы, доступные в классе `Collection`, также доступны в классе `LazyCollection`. Оба эти класса реализуют контракт `Illuminate\Support\Enumerable`, который определяет следующие методы:

- all
- average
- avg
- chunk
- chunkWhile
- collapse
- collect
- combine
- concat
- contains
- containsStrict
- count
- countBy
- crossJoin
- dd
- diff
- diffAssoc
- diffKeys
- dump
- duplicates
- duplicatesStrict
- each
- eachSpread
- every
- except
- filter
- first
- firstOrFail
- firstWhere
- flatMap
- flatten
- flip
- forPage
- get
- groupBy
- has
- implode
- intersect
- intersectByKeys
- isEmpty

- isNotEmpty
- join
- keyBy
- keys
- last
- macro
- make
- map
- mapInto
- mapSpread
- mapToGroups
- mapWithKeys
- max
- median
- merge
- mergeRecursive
- min
- mode
- nth
- only
- pad
- partition
- pipe
- pluck
- random
- reduce
- reject
- replace
- replaceRecursive
- reverse
- search
- shuffle
- skip
- slice
- sole
- some
- sort
- sortBy
- sortByDesc
- sortKeys

- sortKeysDesc
- split
- sum
- take
- tap
- times
- toArray
- toJson
- union
- unique
- uniqueStrict
- unless
- unlessEmpty
- unlessNotEmpty
- unwrap
- values
- when
- whenEmpty
- whenNotEmpty
- where
- whereStrict
- whereBetween
- whereIn
- whereInStrict
- whereInstanceOf
- whereNotBetween
- whereNotIn
- whereNotInStrict
- wrap
- zip

> {note} Методы, которые изменяют коллекцию (такие как `shift`, `pop`, `prepend` и т.д.), **недоступны** в классе `LazyCollection`.

Методы отложенных коллекций

В дополнение к методам, определенным в контракте `Enumerable`, класс `LazyCollection` содержит следующие методы:

`takeUntilTimeout()`

Метод `takeUntilTimeout` возвращает новую отложенную коллекцию, которая будет перечислять значения до указанного времени. По истечении этого времени коллекция перестанет перечислять:

```
$lazyCollection = LazyCollection::times(INF)
    ->takeUntilTimeout(now()->addMinute());

$lazyCollection->each(function ($number) {
    dump($number);

    sleep(1);
});

// 1
// 2
// ...
// 58
// 59
```

Чтобы проиллюстрировать использование этого метода, представьте приложение, которое отправляет счета из базы данных с помощью курсора. Вы можете определить запланированную задачу, которая запускается каждые 15 минут и обрабатывает счета максимум 14 минут:

```
use App\Models\Invoice;
use Illuminate\Support\Carbon;

Invoice::pending()->cursor()
    ->takeUntilTimeout(
        Carbon::createFromTimestamp(LARAVEL_START)->add(14, 'minutes')
    )
    ->each(fn ($invoice) => $invoice->submit());
```

`tapEach()`

В то время как метод `each` вызывает переданное замыкание для каждого элемента в коллекции сразу же, метод `tapEach` вызывает переданное замыкание только тогда, когда элементы извлекаются из списка один за другим:

```
// Пока ничего не выведено ...
$lazyCollection = LazyCollection::times(INF)->tapEach(function ($value) {
    dump($value);
});

// Три элемента выведено ...
$array = $lazyCollection->take(3)->all();

// 1
// 2
// 3
```

`remember()`

Метод `remember` возвращает новую отложенную коллекцию, которая запоминает любые значения, которые уже были перечислены, и не будет извлекать их снова при последующих перечислениях коллекции:

```
// Запрос еще не выполнен ...
$users = User::cursor()->remember();

// Запрос выполнен ...
// Первые 5 пользователей из базы данных включены в результирующий набор ...
$users->take(5)->all();

// Первые 5 пользователей пришли из кеша коллекции ...
// Остальные из базы данных включены в результирующий набор ...
$users->take(20)->all();
```